UCRL-CONF-213753

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

OSKI: A Library of Automatically Tuned Sparse Matrix Kernels

R. Vuduc, J. W. Demmel, K. A. Yelick

July 20, 2005

SciDAC 2005 Proceedings (Journal of Physics)
San Francisco, CA, United States
June 26, 2005 through June 30, 2005

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

OSKI: A library of automatically tuned sparse matrix kernels

Richard Vuduc[1], **James W Demmel**[2], **and Katherine A Yelick**[3]

[1] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-365, Livermore, California 94550, USA

[2] Department of Electrical Engineering and Computer Sciences, and Department of Mathematics, University of California, Berkeley, 737 Soda Hall, Berkeley, California 94720, USA

[3] Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 776 Soda Hall, Berkeley, California 94720, USA

E-mail: `richie@llnl.gov`[1], `{demmel,yelick}@eecs.berkeley.edu`[2,3]

Abstract. The Optimized Sparse Kernel Interface (OSKI) is a collection of low-level primitives that provide *automatically tuned* computational kernels on sparse matrices, for use by solver libraries and applications. These kernels include sparse matrix-vector multiply and sparse triangular solve, among others. The primary aim of this interface is to hide the complex decision-making process needed to tune the performance of a kernel implementation for a particular user's sparse matrix and machine, while also exposing the steps and potentially non-trivial costs of tuning at run-time. This paper provides an overview of OSKI, which is based on our research on automatically tuned sparse kernels for modern cache-based superscalar machines.

1. Goals and Motivation

We describe the Optimized Sparse Kernel Interface (OSKI), a collection of low-level primitives that provide automatically tuned computational kernels on sparse matrices, for use by solver libraries and applications. The kernels include sparse matrix-vector multiply (SpMV) and sparse triangular solve (SpTS), among others; "tuning" refers to the process of selecting the data structure and code transformations that lead to the fastest implementation of a kernel, given a machine and matrix. While conventional implementations of SpMV have historically run at 10% of machine peak or less, careful tuning can achieve up to 31% of peak and $4\times$ speedups [1, Chap. 1]. The challenge is that we must often defer tuning until run-time, since the matrix may be unknown until then. The need for run-time tuning differs from the case of dense kernels where only install- or compile-time tuning has proved sufficient in practice [2, 3].

OSKI reflects the need for and cost of run-time tuning, as extensively documented in our recent work on automatic tuning of sparse kernels using the Sparsity framework [4, 5, 1, 6, 7, 8, 9, 10, 11]. The 6 goals of our interface and the key findings motivating each are as follows:

(i) **Provide basic sparse kernel "building blocks"**: We define an interface for basic sparse operations like SpMV and SpTS, in the spirit of the widely-used Basic Linear Algebra Subroutines (BLAS) [12] and recent Sparse BLAS Standard [13, 12]. We choose the performance-critical kernels needed by sparse solver libraries and applications (particularly those based on iterative solution methods). We target "users" who are sparse solver library writers, or otherwise interested in performance-aware programming at the level of the BLAS.

(ii) **Hide the complex process of tuning**: Matrices in our interface are represented by handles, thereby enabling the library to choose the data structure. We use this indirection because the best data structure and code transformations on modern hardware may be difficult to determine, even in seemingly simple cases [4, 1].
For example, since many sparse matrices have a natural block structure, we can enhance the spatial and temporal locality of SpMV by storing the matrix as a collection of blocks. However, we have observed cases in which SpMV on a matrix with an "obvious" block structure nevertheless runs in 38% of the time of a conventional implementation (2.6× speedup) using a different, non-obvious block structure [1]. Furthermore, we have shown that if a matrix has no obvious block structure, SpMV can still execute in half the time (2× speedup) of a conventional implementation by *imposing* block structure through explicitly stored zeros, even though doing so results in extra work (flops) [1].

(iii) **Offer higher-level memory hierarchy-friendly kernels**: The kernels defined in our interface are a superset of those available in similar library interfaces, including the Sparse BLAS standard [13, 14] and the SPARSKIT library [15], among others [16]. Our "higher-level" kernels are designed for cache-based machines and can execute much faster than their equivalent implementations using "standard" kernels.
For example, in addition to the SpMV operation $y \leftarrow A \cdot x$, we include the kernel $y \leftarrow A^T A \cdot x$ in which A may be read from main memory only once. Compared to a register-blocked two-step implementation, $t \leftarrow A \cdot x, y \leftarrow A^T \cdot t$, a cache-interleaved implementation can be up to 1.8× faster, and up to 4.2× faster than an unblocked two-step implementation [8].

(iv) **Expose the cost of tuning**: We require the user to request tuning explicitly because of its potential costs. In the case of SpMV, tuning can cost 40× as much as a single SpMV operation [1]. Although this cost is dominated by the time simply to copy the matrix to the new data structure (and so is comparable to just building the matrix in the first place), it should nevertheless only be done when the user expects sufficiently many SpMV calls to amortize this cost, as might be expected in iterative solvers [1].

(v) **Support self-profiling**: The user cannot always *a priori* predict, say, the number of SpMV operations that will occur during an application run. We designed our interface to allow the library to monitor transparently all operations performed on a given matrix, and then use this information in deciding how aggressively to tune. Self-profiling enables the library to guess whether tuning will be profitable (see Section 2.3).

(vi) **Allow for user inspection and control of the tuning process**: To help the user reduce the cost of tuning, the interface provides two mechanisms that allow her both to guide and to see the results of the tuning process (Section 3). First, the user may provide explicit hints about the workload (*e.g.*, the number of SpMVs) and the kind of structure she believes the matrix possesses (*e.g.*, uniform blocks of size 3×3, or diagonals). Second, the user may retrieve string-based summaries of what tuning transformations and other performance optimizations have been applied to a given matrix. Thus, a user may see and save these results for re-application on future problems (matrices) which the user believes have similar structure to a previously tuned matrix. Moreover, a user may select and apply transformations manually.

An implementation of OSKI is available [17]. The remainder of this paper highlights features of OSKI by example, and the interested reader may consult the OSKI 1.0 User's Guide for details. We use a library-based approach because it enables the use of run-time information, and because of its potential immediate impact on applications. OSKI could be integrated readily into popular solver libraries such as PETSc [18, 19], or environments such as MATLAB [20, 21].

Listing 1. A usage example without tuning.

```
// This example computes $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$, where
// $A = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ .5 & 0 & 1 \end{pmatrix}$, $x = \begin{pmatrix} .25 \\ .45 \\ .65 \end{pmatrix}$, and $y$ is initially $\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$
// $A$ is a sparse lower triangular matrix with a unit diagonal, and $x, y$ are dense vectors.

// User's initial matrix and data
#define DIM 3 // matrix dimension
#define NNZ_STORED // no. of stored non-zeros
int Aptr[DIM] = {0, 0, 1, 2}, Aind[NNZ_STORED] = {0, 0};
double Aval[NNZ_STORED] = {−2, 0.5};
double x[DIM] = {.25, .45, .65}, y[DIM] = {1, 1, 1};
double alpha = −1, beta = 1;

// Create a tunable sparse matrix object.
oski_matrix_t A_tunable = oski_CreateMatCSR
    (Aptr, Aind, Aval, DIM, DIM, // CSR arrays
    SHARE_INPUTMAT,       // "copy mode"
    // remaining args specify how to interpret non-zero pattern
    3, INDEX_ZERO_BASED, MAT_TRI_LOWER, MAT_UNIT_DIAG_IMPLICIT);

// Create wrappers around the dense vectors.
oski_vecview_t x_view = oski_CreateVecView (x, DIM, STRIDE_UNIT);
oski_vecview_t y_view = oski_CreateVecView (y, DIM, STRIDE_UNIT);

// Perform matrix vector multiply, $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$.
oski_MatMult (A_tunable, OP_NORMAL, alpha, x_view, beta, y_view);

// Clean-up interface objects
oski_DestroyMat (A_tunable);
oski_DestroyVecView (x_view); oski_DestroyVecView (y_view);

// Print result, y. Should be "[ .75 ; 1.05 ; .225 ]"
printf ("Answer: y = [ %f ; %f ; %f ]\n", y[0], y[1], y[2]);
```

2. An Introduction to the Tuning Interface by Example

This section introduces the C version[1] of OSKI using several examples. OSKI uses an object-oriented calling style, where the two main object types are (1) a sparse matrix object, and (2) a dense (multiple) vector object. We anticipate that users will use the library in different ways, so this section illustrates the library's major design points by discussing three such ways.

2.1. Basic usage: gradually migrating applications

To ease the development effort for existing applications, OSKI supports matrix data sharing when the user's sparse matrix starts in a standard array implementation of some basic sparse matrix format, *e.g.*, compressed sparse row (CSR) or column (CSC) formats. Furthermore, users do not have to use any of the automatic tuning facilities, or may introduce the use of tuned operations gradually over time.

[1] Fortran interfaces are under development.

oski_MatMult	Sparse matrix-vector multiply (SpMV) $y \leftarrow \alpha \cdot \mathrm{op}(A) \cdot x$ where $\mathrm{op}(A) \in \{A, A^T, A^H\}$.
oski_MatTrisolve	Sparse triangular solve (SpTS) $x \leftarrow \alpha \cdot \mathrm{op}(A)^{-1} \cdot x$
oski_MatTransMatMult	$y \leftarrow \alpha \cdot \mathrm{op}_2(A) \cdot x + \beta \cdot y$ where $\mathrm{op}_2(A) \in \{A^T A, A^H A, AA^T, AA^H\}$
oski_MatMultAndMatTransMult	Simultaneous computation of $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$, and $z \leftarrow \omega \cdot \mathrm{op}(A) \cdot w + \zeta \cdot z$
oski_MatPowMult	Matrix power multiplication Computes $y \leftarrow \alpha \cdot \mathrm{op}(A)^{\rho} \cdot x + \beta \cdot y$

Table 1. Sparse kernels available in OSKI.

The key feature of OSKI shown in the example of Listing 1 is that OSKI expects "standard representations" of the user's sparse matrix and dense vectors, to minimize changes to existing applications. Listing 1 uses OSKI to compute one SpMV *without* any tuning. The input matrix is 3×3 lower triangular with all ones on the diagonal. The matrix is declared statically in lines 6–9 and stored in CSR format using 2 integer arrays, Aptr and Aind, to represent the non-zero pattern and one array of doubles, Aval, to store the non-zero values. This example assumes 0-based indices and does not store the diagonal explicitly, and this overall representation is a common way of implementing CSR in various sparse libraries [15, 22, 19]. Line 10 declares and initializes two arrays, x and y, to represent the vectors. Like the matrix, these vector declarations are "standard" implementations that the user could pass directly to, say, the dense BLAS to perform dot products or scalar-times-vector products.

Listing 1 creates a tunable matrix object from the input matrix by calling **oski_CreateMat-CSR** (line 14). The arguments specify the untuned physical representation (arguments 1–5 in line 16), the semantics of how to interpret this representation (arguments 7–10 in line 18), and a *copy mode* (argument 6 in line 16) that controls the number of copies of the assembled matrix that may exist at any point in time. The matrix creation routine is the most complex of all of the available OSKI routines, but it supports the specification of a variety of input matrices, as documented fully in the User's Guide, whether the input matrix be symmetric/Hermitian, triangular, use 0- or 1-based indices.

Similarly, dense vector objects are wrappers, or *views*, around user arrays (lines 21–22). A vector view encapsulates basic information about an array, such as its length, or such as the stride between consecutive elements of the vector within the array. As with the BLAS, a non-unit stride allows a dense vector to be a submatrix. In addition, an object of type oski_vecview_t can encapsulate multiple vectors (*multivector*) for kernels like sparse matrix-multiple vector multiply (SpMM) or triangular solve with multiple simultaneous right-hand sides; a blocked SpMM can be up to 2.5× faster than a blocked SpMV [4, 23]. The multivector object would also store the number of vectors and the memory organization (*i.e.*, row vs. column major). Requiring the user to create a view in both the single- and multiple-vector cases helps unify and simplify some of the kernel argument lists, in addition to the potential performance improvements.

The argument lists to kernels, such as **oski_MatMult** for SpMV in this example (line 25), follow some of the conventions of the dense BLAS. For example, a user can specify the constant OP_TRANS as the second argument to apply A^T instead of A, or specify other values for α and β. The list of available kernels appears in Table 1. Beyond SpMV and SpTS, this list includes higher-level kernels designed to exploit memory hierarchies.

Listing 2. An example of basic explicit tuning.

```
1  // Create a tunable sparse matrix object.
2  A_tunable = oski_CreateMatCSR (...);
3
4  // Tell the library we expect to perform 500 SpMV operations with α = 1, β = 1.
5  oski_SetHintMatMult (A_tunable, OP_NORMAL, 1.0, SYMBOLIC_VEC,
6      1.0, SYMBOLIC_VEC, 500);
7  oski_SetHint (A_tunable, HINT_SINGLE_BLOCKSIZE, 6, 6);
8  oski_TuneMat (A_tunable);
9  // ...
10 x_view = oski_CreateVecView (...);
11 y_view = oski_CreateVecView (...);
12
13 for (i = 0; i < 100; i++) {
14     // ...
15     for (k = 0; k < 5; k++) {
16         // ...
17         oski_MatMult (A_tunable, OP_NORMAL, 1.0, x_view, 1.0, y_view);
18         // ...
19     }
20     // ...
21 }
```

That A_tunable, x_view, and y_view are shared with the library implies the user can continue to operate on the data to which these views point as she normally would. For instance, the user can call dense BLAS operations, such as a dot products or scalar-vector multiply, on x and y. Moreover, the user might choose to introduce calls to the OSKI kernels selectively over time.

2.2. Providing explicit tuning hints

Any information the user can provide *a priori* is information the library does not need to rediscover, thereby reducing the overhead of tuning. In this case, a user may provide the library with structured hints to describe, for example, the expected workload (*i.e.*, which kernels will be used and how frequently), or whether there is special non-zero structure (*e.g.*, uniformly aligned dense blocks, symmetry). The user then calls a special "tune routine" to choose a new data structure performance-optimized for the specified workload. We refer to this style of OSKI usage as *tuning with explicit hints*.

Listing 2 shows how to provide hints. The first hint (lines 5–6) specifies the expected workload will consist of at least a total of 500 SpMV operations on the same matrix. The argument list is identical to the corresponding argument list for the kernel call, **oski_MatMult**, except that there is one additional parameter to specify the expected frequency of SpMV operations. The frequency allows the library to decide whether there are enough SpMV operations to hide the cost of tuning. For optimal tuning, the values of these parameters should match the actual calls as closely as possible.

The constant SYMBOLIC_VEC indicates that we will apply the matrix to a single vector with unit stride. Alternatively, we could use the constant SYMBOLIC_MULTIVEC to indicate that we will perform sparse SpMM on at least two vectors. Better still, we could pass an actual instance of a oski_vecview_t object which has the precise stride and data layout information. Analagous routines exist for each of the other kernels in the system.

The second hint (line 7) is a *structural hint* indicating that the matrix non-zero structure *may*

Listing 3. An example of implicit tuning.

```
 1  oski_matrix_t A_tunable = oski_CreateMatCSR (...);
 2  oski_vecview_t x_view = oski_CreateVecView (...);
 3  oski_vecview_t y_view = oski_CreateVecView (...);
 4  oski_SetHint (A_tunable, HINT_SINGLE_BLOCKSIZE, 6, 6);
 5  // ...
 6  for (i = 0; i < num_times; i++) {
 7      // ...
 8      while (!converged) {
 9          // ...
10          oski_MatMult (A_tunable, OP_NORMAL, 1.0, x_view, 1.0, y_view);
11          // ...
12      }
13      oski_TuneMat (A_tunable);
14      // ...
15  }
```

be dominated by 6×6 dense subblocks. Several of the possible structural hints accept optional arguments that may be used to qualify the hint. The hints currently available are related to candidate optimizations explored in our work, and the list of hints will grow over time.

The actual tuning (*i.e.*, possible change in data structure) occurs at the call to **oski_TuneMat**. The OSKI library uses all hint information provided up to this call to tune, *i.e.*, select a new data structure which is likely to improve performance for the specified matrix and workload. This new data structure is only used internally by OSKI at kernel calls, and so does not affect the user's original input matrix data structure. Note that the call to **oski_TuneMat** marks the point during program execution at which tuning (and therefore, its overhead) may occur, thereby exposing the tuning step.

2.3. Tuning based on implicit profiling

The library needs a workload to decide when the overhead of tuning can be amortized, but the user cannot always estimate this workload before execution as done in Section 2.2. In OSKI, a user may instead rely on the library to monitor kernel calls to determine the workload dynamically. The user must still call **oski_TuneMat** to tune, but this call optimizes based on a workload inferred from the kernel calls executed so far. Listing 3 provides no workload hints, but calls **oski_TuneMat** periodically (line 13). Internally, the library can monitor the calls to **oski_MatMult**, and at each call to **oski_TuneMat** evaluate whether there seem to be enough SpMV calls to hide the tuning cost.

3. Saving and Restoring Tuning Transformations

To promote transparency in the tuning process, OSKI allows the user to see a precise description, represented by a string, of the transformations that create the tuned data structure. In OSKI 1.0, this string is a program expressed in a procedural, high-level scripting language, OSKI-Lua (derived from the Lua language [24]). The user may subsequently "execute" this program on the same or similar input matrix, thereby providing a way to save and restore tuning transformations across application runs, *a la* FFTW's *wisdom* mechanism [25]. Moreover, this mechanism allows an advanced user to specify her own sequence of optimizing transformations, and allows OSKI developers to extend OSKI to include new techniques.

Listing 4 contains a code fragment which reads a transformation string (OSKI-Lua program) from a file, and applies it to a matrix using OSKI's **oski_ApplyMatTransforms**. This call

Listing 4. An example of applying transformations.

```
FILE* fp_saved_xforms = fopen ("./my_xform.txt", "rt"); // file containing transformation to apply
oski_matrix_t A_tunable = oski_CreateMat_CSR (...);
char xforms[MAXBUFSIZE]; // transform to apply
fread (xforms, ..., fp_saved_xforms); // read transform from file
oski_MatMult (A_tunable, ...); // untuned SpMV
oski_ApplyMatTransforms (A_tunable, xforms); // change data structure
oski_MatMult (A_tunable, ...); // tuned SpMV
```

is equivalent to calling **oski_TuneMat**, except that instead of allowing the library to decide what data structure to use, we are specifying it explicitly. (OSKI has an analogous routine, **oski_GetMatTransforms**, to retrieve the last transformation applied to a matrix.)

The syntax of OSKI-Lua is based on the Lua scripting language [24]. The following example computes a structural splitting, $A = A_1 + A_2 + A_3$, where A_1 is stored in 4×2 UBCSR, A_2 is stored in 2×2 UBCSR, and A_3 is stored in CSR(comments preceeded by #). This example uses VBRas an intermediate format for determining how to split.

```
1    # Let $A = A_1 + A_2 + A_3$, where $A_1$ is in $4 \times 2$ UBCSR,
2    # $A_2$ is in $2 \times 2$ UBCSR, and $A_3$ is in CSR
3    T = VBR(InputMat);
4    # First, split $A = A_1 + A_{\text{leftover}}$, where $A_{\text{leftover}}$ is in CSRformat
5    A1, A_leftover = T.extract_blocks(4, 2);
6    # Next, split $A_{\text{leftover}} = A_2 + A_3$
7    T = VBR(A_leftover);
8    A2, A3 = T.extract_blocks(2, 2);
9    return A1 + A2 + A3;
```

(Structural splitting in this style yields speedups of up to 1.8× over a blocked but not split implementation [26].)

When **oski_ApplyMatTransforms** executes, it interprets the program to carry out the transformation. The last line executed by every OSKI-Lua program returns the new data structure—here, the union of the split components A_1, A_2, and A_3, represented by the symbolic summation in line 10. Garbage collection of temporaries is performed automatically.

4. Other Features

OSKI provides functionality beyond the core kernels and tuning facilities:

- **Support for various scalar precisions**: Like the BLAS, non-zero values may be real or complex, single- or double-precision. In addition, the integer indices associated with the sparse matrix may be C `int` or `long`.
- **Changing/retrieving matrix entries**: As long as the *pattern* remains fixed, the user may change the values of any structural non-zero entries using OSKI's get/set value routines.
- **Explicit representation of permutations**: During tuning, the library may decide to permute the rows and columns of the matrix to improve locality, but to maintain correctness it must permute input/output vectors at every kernel call. OSKI allows the user to detect, extract, and apply these permutations herself if her algorithm can permute less frequently.
- **Two error handling styles**: Users can check the return value of any OSKI call for errors. In addition, the user may supply her own error handler for logging or recovery purposes.

These features are described in complete detail in the OSKI 1.0 User's Guide.

5. Related Work: Approaches that Complement Libraries

The Sparse BLAS Standard [13] inspired the OSKI design. The main differences are (i) we do not specify primitives for matrix construction, and instead assume that the user can provide an assembled matrix in one of a few formats, and (ii) we include explicit support for tuning.

A number of approaches complement the library approach. One is to implement a library using a language with generic programming constructs such as templates in C++ [27]. Both Blitz++ [28] and the Matrix Template Library (MTL) [29] have adopted this approach to building generic libraries in C++ that mimic dense BLAS functionality. Templates faciliate the generation of large numbers of routines from a compact representation, and flexibly handles issues of producing libraries that can handle different precisions. Sophisticated use of templates (template metaprogramming) also allows some optimization, such as unrolling and some forms of loop fusion [28]. However, this approach does not address run-time tuning.

Compiler-based sparse code generation, via restructuring compilers, extends the generic programming idea (Bik [30, 31, 32], Stodghill, *et al* [33, 34, 35, 36], and Pugh and Shpeisman [37, 38]). These are clean, general approaches to code generation. The user expresses separately both the kernels (as dense code with random access to matrix elements) and a specification of the sparse data structure; a restructuring compiler combines the two descriptions to produce a sparse implementation. Since any kernel can in principle be expressed, this overcomes a library approach in which all possible kernels must be pre-defined. We view this technology as complementary to the overall library approach; while sparse compilers could be used to provide the underlying implementations of sparse primitives, they do not explicitly make use of matrix structural information available, in general, only at run-time.

A third approach is to extend an existing library or system. There are a number of application-level libraries (*e.g.*, PETSc [18, 19], among others [39, 15, 22, 40]) and high-level application tools (*e.g.*, MATLAB [20, 21], Octave [41], approaches that apply compiler analyses and transformations to MATLAB code [42, 43]) that provide high-level sparse kernel support. Integration with these systems, which have large user bases, allows complete hiding of data structure details and the tuning process from the user. The goal of OSKI is to provide building blocks in the spirit of the BLAS with the steps and costs of tuning exposed. It should be possible to integrate the OSKI library into an existing system as well, as has been done successfully with the integration of ATLAS and FFTW tuning systems into MATLAB.

6. Conclusions and Future Work

OSKI is designed to provide a migration path for existing solver libraries and applications to use high-performance implementations of sparse matrix kernels. An OSKI user whose sparse matrix is already available pre-assembled in standard CSR or CSC array representations can introduce calls to OSKI's kernels selectively and gradually over time. Tuning is automatic, but only occurs when the user explicitly requests it, thus allowing the user to decide when it is most appropriate to tune in her application.

The OSKI 1.0 implementation [17] is a uniprocessor library that targets cache-based superscalar machines. We are actively pursuing shared and distributed memory versions of OSKI, as well as tuning specific to alternative architectures such as vector processor-based machines. OSKI-based applications will be performance-portable on such platforms.

In addition, OSKI has been released as open-source. The implementation is modular, so that new matrix formats and tuning heuristics can be added to an existing OSKI installation without recompiling the core library, or even the application on systems with shared library support. We are making our own sparse kernel tuning research [44, Sec. 4.3] available as modules in OSKI.

To reach still larger user communities, we are also implementing an OSKI matrix type for PETSc. The OSKI 1.0 User's Guide outlines how we envision OSKI being implemented into other libraries, including the Sparse BLAS [13] and MATLAB*P [45].

References

[1] Richard Vuduc. *Automatic performance tuning of sparse matrix kernels.* PhD thesis, University of California, Berkeley, Berkeley, CA, USA, December 2003.

[2] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC.

[3] R. Clint Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.

[4] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[5] Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[6] Benjamin C. Lee, Richard Vuduc, James W. Demmel, Katherine A. Yelick, Michael deLorimier, and Lijue Zhong. Performance optimizations and bounds for sparse symmetric matrix-multiple vector multiply. Technical Report UCB/CSD-03-1297, University of California, Berkeley, Berkeley, CA, USA, November 2003.

[7] Rajesh Nishtala, Richard Vuduc, James Demmel, and Katherine Yelick. When cache blocking sparse matrix vector multiply works and why. In *Proceedings of the PARA'04 Workshop on the State-of-the-art in Scientific Computing*, Copenhagen, Denmark, June 2004.

[8] Richard Vuduc, Attila Gyulassy, James W. Demmel, and Katherine A. Yelick. Memory hierarchy optimizations and bounds for sparse $A^T Ax$. In *Proceedings of the ICCS Workshop on Parallel Linear Algebra*, volume LNCS, Melbourne, Australia, June 2003. Springer.

[9] Richard Vuduc, Shoaib Kamil, Jen Hsu, Rajesh Nishtala, James W. Demmel, and Katherine A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries*, New York, USA, June 2002.

[10] Christopher Hsu. Effects of block size on the block Lanczos algorithm, June 2003. Senior thesis.

[11] Berkeley Benchmarking and OPtimization (BeBOP) Project, 2004. `bebop.cs.berkeley.edu`.

[12] Susan L. Blackford, James W. Demmel, Jack Dongarra, Iain S. Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[13] Iain S. Duff, Michael A. Heroux, and Roland Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software*, 28(2):239–267, June 2002.

[14] Iain S. Duff and Christof Vömel. Algorithm 818: A reference model implementation of the sparse BLAS in Fortran 95. *ACM Transactions on Mathematical Software*, 28(2):268–283, June 2002.

[15] Yousef Saad. SPARSKIT: A basic toolkit for sparse matrix computations, 1994. `www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`.

[16] Salvatore Filippone and Michele Colajanni. PSBLAS: A library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software*, 26(4):527–550, December 2000.

[17] Richard Vuduc, James Demmel, and Katherine Yelick. OSKI: An interface for a self-optimizing library of sparse matrix kernels, 2005. `bebop.cs.berkeley.edu/oski`.

[18] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[19] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matt Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc User's Manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2002. www.mcs.anl.gov/petsc.

[20] Matlab, 2003. The MathWorks, Inc. `www.mathworks.com`.

[21] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[22] K. Remington and R. Pozo. NIST Sparse BLAS: User's Guide. Technical report, NIST, 1996. `gams.nist.gov/spblas`.

[23] Benjamin C. Lee, Richard Vuduc, James Demmel, and Katherine Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *Proceedings of the International Conference on Parallel Processing*, Montreal, Canada, August 2004.

[24] Roberto Ierusalimschy, Luiz Henrique de Figeiredo, and Waldemar Celes. Lua 5.0 Reference Manual.

Technical Report MCC-14/03, PUC-Rio, April 2003. www.lua.org.

[25] Matteo Frigo and Stephen Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Seattle, Washington, May 1998.

[26] Richard Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. Technical Report UCRL-TR-213454, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA, July 2005.

[27] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software: Practice and Experience*, 24:632–642, 1994.

[28] Todd Veldhuizen. Arrays in Blitz++. In *Proceedings of ISCOPE*, volume 1505 of *LNCS*. Springer-Verlag, 1998.

[29] Jeremy G. Siek and Andrew Lumsdaine. A rational approach to portable high performance: the Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (fast) library. In *Proceedings of ECOOP*, Brussels, Belgium, 1998.

[30] Aart Johannes Casimir Bik. *Compiler Support for Sparse Matrix Codes*. PhD thesis, Leiden University, 1996.

[31] Aart J. C. Bik, Peter J. H. Birkhaus, Peter M. W. Knijnenburg, and Harry A. G. Wijshoff. The automatic generation of sparse primitives. *ACM TOMS*, 24(2):190–225, July 1998.

[32] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic nonzero structure analysis. *SIAM Journal on Computing*, 28(5):1576–1587, 1999.

[33] Paul Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. PhD thesis, Cornell University, August 1997.

[34] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill. A framework for sparse matrix code synthesis from high-level specifications. In *Proceedings of Supercomputing 2000*, Dallas, TX, November 2000.

[35] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *International Conference on Supercomputing*, 2000.

[36] Nikolay Mateev, Keshav Pingali, and Paul Stodghill. The Bernoulli Generic Matrix Library. Technical Report TR-2000-1808, Cornell University, 2000.

[37] William Pugh and Tatiana Shpeisman. Generation of efficient code for sparse matrix computations. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, LNCS, August 1998.

[38] J. Irwin, J.-M. Loingtier, John Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proceedings of the International Scientific Computing in Object-Oriented Parallel Environments*, Marina del Rey, CA, USA, December 1997.

[39] Alan George and Joseph W. H. Liu. The design of a user interface for a sparse matrix package. *ACM Transactions on Mathematical Software*, 5(2):139–162, June 1979.

[40] Bjørn-Ove Heimsund. JMP: A sparse matrix library in Java, 2003. http://www.mi.uib.no/∼bjornoh/jmp.

[41] John W. Eaton. Octave, 2003. www.octave.org.

[42] George Almási and David Padua. MaJIC: Compiling MATLAB for speed and responsiveness. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[43] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, Austin, TX, October 1999.

[44] James Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Richard Vuduc, R. Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. In *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation*, February 2005.

[45] Viral Shah and John R. Gilbert. Sparse matrices in Matlab *P: Design and implementation. In *Proceedings of the International Conference on High-Performance Computing*, volume 3296 of *LNCS*, pages 144–155, Bangalore, India, 2004. Springer.

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.